Regular Expressions and Gender Guessing

This tour-de-force AWK program reveals the strengths and weaknesses of the rule-based paradigm and will prove useful for mailing-list programmers.

Scott Pakin

Suppose you were given a long mailing list and had to write a program to determine whether each name on the list represented a male or female, perhaps so you could preface each name with "Mr." or "Ms." in the salutation of a form letter. How would you do it?

The first solution that comes to mind might be to create a table of every personal name and the gender normally associated with it. Then, for each name in the mailing list, look up the corresponding gender in the table, most likely using a hash function for speed.

Such an approach has two key problems. First, variations in spelling, no matter how slight, will defeat the algorithm. A human can tell that Caren, Caryn, Karyn, and Karin are all variations of the same name, but a table-lookup algorithm can't. If one of those four spellings aren't in the table, the program will be unable to match that name with a gender. Second, typing thousands of name-gender pairs into a table is time-consuming and error-prone. Moreover, you might never have a complete list because of variant spellings.

What if I told you a gender-detection program can be written in about 50 lines of code and data combined? Sound too good to be true? In fact, not only is such a routine entirely feasible, but I have written one myself. This program demonstrates the power of AWK and rule-based programming. It also points up some weaknesses of the rule-based approach.

Who's who?

To start, I looked for a new approach to solving the problem by examining the input set: people's first names. Names are not random sequences of characters, but rather are formed following certain rules. By determining what some of the rules are, you can program some smarts into a lookup routine.

In other words, instead of matching straight text, the program matches based on patterns within the text. The strategy says that if a name matches a certain pattern of letters, it must be a certain gender. For example, if a name contains the letter sequence *ann* (upper or lowercase), it must be female (not including foreign names, diminutives of English names, and ambiguous names). This rule catches a lot of names, including Ann, Anna, Annabelle, Annacarol, Annalisa, Annaliz, Annamarie, Anne, Annemarie, Annette, Annie, Arianna, Deanne, Diann, Georgann, Georgieanne, Hannah, Jeanne, Jeannette, Joann, Joanna, Joanne, Juliann, Leann, Lewanna, Lianna, Louann, Marianne, Maryann, Maryanne, Rosanne, Roseanne, Ruthann, Ruthanna, Sallyann, Sherrianne, Susanna, Susanne, and Suzanne. Whew!

A number of advantages are associated with using a pattern-matching approach, even beyond the amount of time saved by not having to enter thousands of names and their associated genders into a table. First, the program will often work with a name or variation of a name not yet encountered. Second, and in a similar vein, typos and misspelled names still tend to yield the same gender as the correct transcription. And third, the program is guaranteed to return a gender for any name given it. You can't stump the code.

LISTING 1.

```
cat /etc/passwd | awk -F: '{print $5}' | awk '{print $1}' | sort | uniq > firstnames
```

LISTING 2.

```
# PREDICT GENDER GIVEN A FIRST NAME
# by Scott Pakin, August 1991

                                              { sex = "m" } # Assume male.

/^.*[aeiy]$/                                  { sex = "f" } # Most names that end
                                              in a/e/i/y are female.
/^All?[iy]((ss?)|z)on$/                       { sex = "f" } # Allison (and variations)
/^.*een$/                                     { sex = "f" } # Cathleen, Eileen,
                                                              Maureen,...
/^[^S].*r[rv]e?y?$/                           { sex = "m" } # Barry, Larry, Perry,...
/^[^G].*v[ei]$/                               { sex = "m" } # Clive, Dave, Steve,...
/^[^BD].*(b[iy]|y|via)nn?$/                   { sex = "f" } # Carolyn, Gwendolyn,
                                                              Vivian,...
/^[^AJKLMNP][^o][^eit]*([glrsw]ey|lie)$/      { sex = "m" } # Dewey, Stanley, Wesley,...
/^[^GKSW].*(th|lv)(e[rt])?$/                  { sex = "f" } # Heather, Ruth, Velvet,...
/^[CGJWZ][^o][^dnt]*y$/                       { sex = "m" } # Gregory, Jeremy,
                                                              Zachary,...
/^.*[Rlr][abo]y$/                             { sex = "m" } # Leroy, Murray, Roy,...
/^[AEHJL].*il.*$/                             { sex = "f" } # Abigail, Jill, Lillian,...
/^.*[Jj](o|o?[ae]a?n.*)$/                     { sex = "f" } # Janet, Jennifer, Joan,...
/^.*[GRguw][ae]y?ne$/                         { sex = "m" } # Duane, Eugene, Rene,...
/^[FLM].*ur(.*[^eotuy])?$/                    { sex = "f" } # Fleur, Lauren, Muriel,...
/^[CLMQTV].*[^dl][in]c.*[ey]$/                { sex = "m" } # Lance, Quincy, Vince,...
/^M[aei]r[^tv].*([^cklnos]|([^o]n))$/         { sex = "f" } # Margaret, Marylou,
                                                              Miriam,...
/^.*[ay][dl]e$/                               { sex = "m" } # Clyde, Kyle, Pascale,...
/^[^o]*ke$/                                   { sex = "m" } # Blake, Luke, Mike,...
/^[CKS]h?(ar[^lst]|ry).+$/                    { sex = "f" } # Carol, Karen, Sharon,...
/^[PR]e?a([^dfju]|qu)*[lm]$/                  { sex = "f" } # Pam, Pearl, Rachel,...
/^.*[Aa]nn.*$/                                { sex = "f" } # Annacarol, Leann,
                                                              Ruthann,...
/^.*[^cio]ag?h$/                              { sex = "f" } # Deborah, Leah, Sarah,...
/^[^EK].*[grsz]h?an(ces)?$/                   { sex = "f" } # Frances, Megan, Susan,...
/^[^P]*([Hh]e|[Ee][lt])[^s]*[ey].*[^t]$/      { sex = "f" } # Ethel, Helen, Gretchen,...
/^[^EL].*o(rg?|sh?)?(e|ua)$/                  { sex = "m" } # George, Joshua, Theodore,..
/^[DP][eo]?[lr].*se$/                         { sex = "f" } # Delores, Doris,
                                                              Precious,...
/^[^JPSWZ].*[denor]n.*y$/                     { sex = "m" } # Anthony, Henry, Rodney,...
/^K[^v]*i.*[mns]$/                            { sex = "f" } # Karin, Kim, Kristin,...
/^Br[aou][cd].*[ey]$/                         { sex = "m" } # Bradley, Brady, Bruce,...
/^[ACGK].*[deinx][^aor]s$/                    { sex = "f" } # Agnes, Alexis, Glynis,...
/^[ILW][aeg][^ir]*e$/                         { sex = "m" } # Ignace, Lee, Wallace,...
/^[^AGW][iu][gl].*[drt]$/                     { sex = "f" } # Juliet, Mildred,
                                                              Millicent,...
/^[ABEIUY][euz]?[blr][aeiy]$/                 { sex = "m" } # Ari, Bela, Ira,...
/^[EGILP][^eu]*i[ds]$/                        { sex = "f" } # Iris, Lois, Phyllis,...
/^[ART][^r]*[dhn]e?y$/                        { sex = "m" } # Randy, Timothy, Tony,...
/^[BHL].*i.*[rtxz]$/                          { sex = "f" } # Beatriz, Bridget,
                                                              Harriet,...
/^.*oi?[mn]e$/                                { sex = "m" } # Antoine, Jerome, Tyrone,...
/^D.*[mnw].*[iy]$/                            { sex = "m" } # Danny, Demetri, Dondi,...
/^[^BG](e[rst]|ha)[^il]*e$/                   { sex = "m" } # Pete, Serge, Shane,...
/^[ADFGIM][^r]*([bg]e[lr]|il|wn)$/            { sex = "f" } # Angel, Gail, Isabel,...

                                              { print $0, sex } # Output prediction
```

In all fairness, a rule-based approach does have its down side. Because the program will find a gender for every name, it doesn't know when it's made a mistake. A text lookup table can say it doesn't know what gender a name is, but a pattern-matcher cannot. Also, no hard-and-fast rule exists to ascertain which patterns are necessary for the program to guess genders accurately. The approach requires some ingenuity on the part of the programmer and a lot of trial and error.

Making the rules

As you know, English is too cumbersome and vague to be unambiguously expressible on a computer. Regular expressions provide the clarity and succinctness needed for computer programs. Using regular expressions, the concept: "If a name ends in any letter except *c*, *i*, or *o*, followed by an *a*, followed by an optional *g*, followed by an *h*, then it's female" can be expressed as `^.*[^cio]ag?h$` → *female.*

Now that you have a notation for discussing patterns, you must decide what makes a rule (a pattern plus an implication) good or bad at guessing your goal (in this case, gender). Here are some general guidelines I use to evaluate rules:

- Keep it simple. Rules should not be excessively complicated. I've found it's a lot easier to debug and modify a series of simple rules than a single incomprehensible one.
- Maximize the number of correct guesses. Rules should correctly identify the gender of as many and as wide a variety of names as possible, without becoming too complicated. For example, a rule like `^[CK]h?r.*l$` → *female,* which correctly identifies only Crystal, Chrystal, Krystal, and Khrystal, is not as good as a rule like `^[CKS]h?(ar[^lst]|ry).+$` → *female,* which catches those four as well as Caren,

Carin, Carina, Carissa, Carmel, Carmella, Carmen, Carmine, Carol, Carole, Caroline, Carolla, Carolyn, Carrie, Carroll, Caryn, Karen, Karin, Karol, Karyn, Sarah, Sarina, Sharon, and Sharyn.

- Minimize incorrect guesses. Rules should incorrectly identify the gender of as few and as narrow a variety of names as possible, without violating the first two rules. It's better for a rule to be too general than too specific—you can always add a second rule to catch the cases the first one missed.
- Avoid rule overlap. No two rules should apply to identical or nearly identical sets of names. Breaking this condition makes your rules harder to judge. You might be misled to think that two rules are good because they each catch 15 when both rules share 12 of those names. Further, when optimizing your program by reducing the number of rules, you might want to remove a rule that has been made obsolete by another rule. If multiple rules apply to the same set of names, it will be difficult to tell which rules can be removed safely.

I hope these guidelines prove somewhat intuitive. The big question, of course, is: Where do the patterns and rules come from? It turns out that trial and error (and lots of it) is the only method. There are no sure-fire tricks.

Establishing the patterns

The first few rules I wrote pertained to the endings of names. Thinking of the first batch of names that came to mind, I noticed that a tremendous number of female names end in *a,* but very few male names do. *i* is another common female ending, and, to a lesser extent, so are *e* and *y*. I made my first rule ∧.*[aeiy]$ → *female.*

Ultimately, I made it my second rule. My first rule became the most general, ∧.*$ → *male,* which assumes the name is male and is a good fall-back when no other rule matches.

How does the *a,e,i,y* rule stand up to the three guidelines I stated previously? It uses a fairly simple pattern, and it gets a lot of names correct. But it also gets a lot of names wrong; not as many as it gets right, but enough to say that it doesn't minimize incorrect guesses. Still, I claim it's a good rule. If you were constructing a lookup table of names, would you have thought to include the names Shobhana, Meta, and Ottolie? Probably not, but that one rule gets all three correct as well as more popular names like Julie, Mary, and Rebecca.

Because my first two rules are wrong so often, the remaining 95% of my program is just a list of exceptions to them. This technique is typical of rule-based systems. Actually, a few rules of the rules are exceptions to the exceptions. I learned later it's best to minimize the number of these cases. When I was first writing the code, most of my rules were exceptions to exceptions.

For instance, I followed ∧.*$ → *male* and ∧.*[aeiy]$ → *female* with ∧.*ke$ → *male* to catch Blake, Ike, Jake, Luke, and Mike. Then I noticed that the rule failed for Brooke, so I added ∧Brooke$ → *female.* Not only does that last line break away from the guidelines, but I found that if you keep tacking exceptions to exceptions sequentially, the program gets harder and harder to debug. You'll always be wondering why you explicitly specified that "Brooke" is female even though it ends in *e* until you recall that Brooke is an exception to an exception to the *a,e,i,y* rule. It's important to stick to the guidelines.

Frequently, you can use a complemented character class to reduce the length of exception chains. In the previous example, I replaced ∧.*ke$ → *male* and ∧Brooke$ → *female* with ∧[∧o].*ke$ → *male.* That correction makes the code a line shorter, eliminates a rule that violates one of the guidelines, and makes the

UNDERSTANDING REGULAR EXPRESSIONS

Regular expressions are a convenient way to describe patterns in text because a single regular expression can represent any number of text strings. You can think of a regular expression as an object that says whether or not a given text string is representable by a certain pattern:

```
text string →  | regular expression |  → TRUE or FALSE
```

Regular expressions are composed of regular ASCII characters. Most characters match only themselves. For instance, the character *G* matches only *G*, not *g* or any other character. A regular expression can be made up of other regular expressions, so *Gw0P* matches any string that contains the characters *G*, *w*, *0*, and *P*, in that order, with no interceding characters.

Not all characters match themselves. Certain characters are called metacharacters and don't (usually) match themselves, but instead have a special interpretation. "Arity" is the number of additional regular expressions that must be present for the expression to form a syntactic unit. The arity 0 metacharacters follow:

- ∧ matches the beginning of a string.
- $ matches the end of a string.
- . matches any character (including itself).
- \ is the escape metacharacter and has the same meaning as it does in C. The following character is to be taken literally. For example, \$ refers to the $ character, not the end of the string. \\ matches the \ character.
- [...] is called a character class and matches any of the characters enclosed within the brackets. For instance, [sdp] matches either *s*, *d*, or *p*. Within a character class, all characters match themselves with the following exceptions: \ is still the escape metcharacter, and –, when between characters, designates a range. For example, [P-S] matches either *P*, *Q*, *R*, or *S*. And ∧, if it appears as the first character after the opening bracket, turns the character class into a complemented character class, which matches any character not enclosed between the brackets. For example, [∧AEIOUaeiou] will match any nonvowel.

The arity 1 metacharacters follow:

- ? means the preceding regular expression is optional. Thus, *cape?s* matches any string containing either *caps* or *capes*. (The ? applies only to the *e*.)
- * is often called the Kleene star, and means the preceding regular expression may be included zero or more times. For example, be*p matches any string containing *bp*, *bep*, *beep*, *beeep*, and so on.
- + is similar to the Kleene star, except the preceding regular expression must occur at least once. So *be+p* matches everything be*p does except *bp*.

The arity 2 metacharacters follow:

- | matches either the regular expression to the left or right, but not both at once. *a|z* matches any string containing either an *a* or a *z*.

In addition to the these metacharacters, parentheses may be used for grouping. Grouping serves the same purpose in regular expressions as it does in arithmetic: altering the order of operations. The normal order of operations for regular expressions is *, +, and ? first, followed by concatenation (sequences of regular expressions), and | last.

Here are some examples of regular expressions:

- ∧[∧-∧.[\]?!]$ matches any one-character string except those containing one of the characters –, ∧, ., [,], ?, or !.
- *good|bad|ugly* matches any string containing *good*, *bad*, or *ugly*; for example, *forbade*.
- ∧(ab*|[cd]+e?)$ matches either *a* followed by zero or more *b*s or one or more *c*s or *d*s (or a combination) followed by an optional *e*.
- ∧All?[iy](ss?|z)on$ matches Alison, Allison, Alyson, Allyson, Alisson, Allisson, Alysson, Allysson, Alizon, Allizon, Alyzon, and Allyzon.

code a bit more maintainable.

It's fairly easy to use AWK, a programming language designed for simple text manipulation and available on almost all UNIX systems, to extract every user's first name from /etc/passwd. The complete UNIX command is shown in Listing 1.

Writing the code

The command takes the /etc /passwd file, extracts the user's full name (the fifth colon-delimited column), extracts the user's first name (the first space-delimited column), sorts the names alphabetically, and eliminates duplicate names. Pretty neat, huh?

If you're planning on using a gender-detection routine for a mailing list and already have a database, it shouldn't be too difficult to extract the first names from it into a text file to run the regular expression searches. It may sound silly, but I found that one of the best ways to test a program is to present your code to your friends and say, "I'll bet you can't find a name that will stump my program." You're sure to get a long list of cases you never would have thought of.

Merely reading about the benefits of a rule-based, gender-detection system isn't as convincing as seeing an actual gender-guessing program and being able to play with it yourself. Listing 2 contains the complete code for the program I've been describing. Weighing in at only about 50 lines, the terseness of a rule-based approach is quite striking.

True, the program doesn't know every name, and it has particular trouble with foreign names and diminutives of English names. I never bothered getting the program to recognize them because I don't know many foreign names, let alone their gender. And mailing lists usually don't contain diminutives anyway.

PROGRAMMING IN AWK

AWK is an easy-to-learn and easy-to-use programming language. A person comfortable with C should be able to master AWK in about an hour. AWK was created in 1977 by Alfred Aho, Peter Weinberger, and Brian Kernighan to perform small, simple tasks with little code and minimal overhead. The assumption AWK makes is that programs tend to read a line from standard input, process it (usually by manipulating its fields), and write some result to standard output. AWK has been optimized for that sort of program.

Most lines in AWK are of the form *pattern* { *action* }, where either *pattern* or *action* (but not both) are optional. AWK interprets each line as: "If pattern is True, perform action." If the pattern is omitted, the action will always be taken. If the action is omitted, the program defaults to print the entire line. Patterns are usually regular expressions or C-style conditional expressions (that is, what follows an *if* or *while* statement). Actions are statements or function calls and follow C syntax, more or less.

AWK programs are executed using the following pseudocode (roughly):

```
repeat
  get a line from standard input
  if pattern1 is true for the current line, perform action1
  if pattern2 is true for the current line, perform action2
                          .
                          .
                          .
  if patternn is true for the current line, perform actionn until no more lines
```

Variables in AWK don't need to, and in fact can't, be declared. Referencing a variable is sufficient to create it. AWK has only two data types: number and string (and arrays of either, with either number or string indices). An extremely convenient feature is that all variables are initialized to zero, the empty string, or an empty array. Actually, there's no difference between the three at the offset because AWK uses full type coercion. If you add a number to a variable, the variable is treated as a number, and if you concatenate a string to it, it's treated as a string. So after executing the action *myvar = 3*6 "AWK"*, the variable *myvar* will contain the string *18AWK*.

Built into AWK is the concept of a data column. By default, columns are whitespace-delimited pieces of text from an input line. The variables *$1*, *$2*, *$3*, and so on, refer to the contents of the first column, the second column, the third column, and so on. *$0* refers to the entire line. Naturally, the contents of the column variables change with every input line.

AWK has a number of built-in variables. The most useful are *NR* (the number of records or lines read so far), *NF* (the number of fields on the current line), and *FS* (the field separator that separates columns of data). AWK also has a couple of built-in patterns: *BEGIN* and *END*. *BEGIN* is true before any input lines have been read and typically used for initialization. *END* is true after the entire file has been processed and typically used for outputting final results.

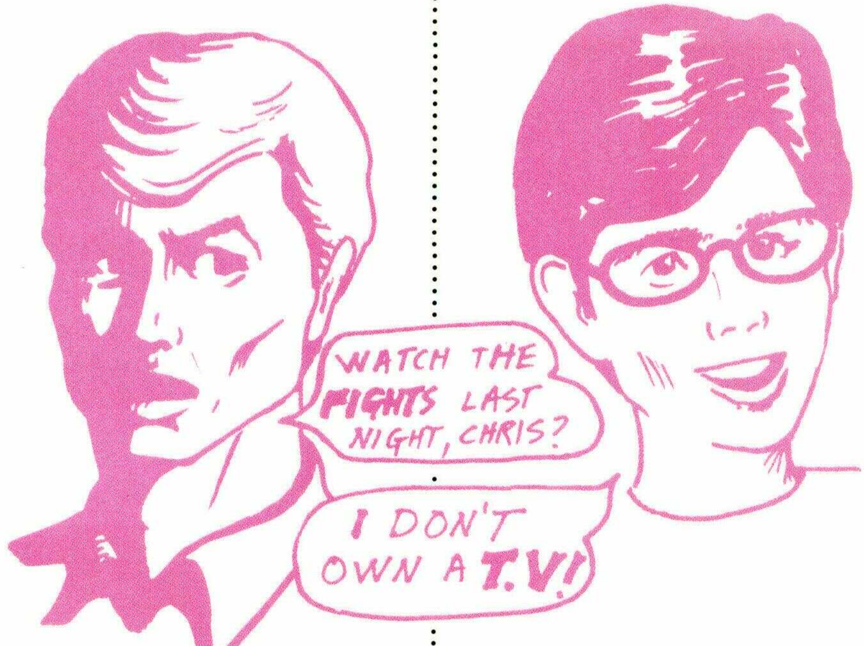
Here are three examples of complete AWK programs:

```
# Output the average of a list of numbers
        { total += $0 }       # Add value of each line to total (as in C)
END     { print total/NR }

# Replace all blank lines with "."
NF==0  { print "." }
NF>0                          # Leave other lines as is

# Print the first 10 characters of the longest line containing the regular
# expression "a.cd?[xyz]*"
/a.cd?[xyz]*/ && \            # \ means continue on next line
length($0)>maxlen     { longline=$0; maxlen=length($0) }
END                   { printf ("%.10s",longline }           # Just like C
```

Remember in Programming 101 when you had to read an input file where each line contained a person's name, hourly wage, and number of hours worked? You had to output another file containing the person's name and how much he or she should get paid. You can write that entire program on the UNIX command line:

```
cat infile | awk '{ print $1,$2*$3 }' > outfile
```

AWK is one of those gems that comes free with almost all UNIX systems and is absolutely terrific for all those tiny programs you don't want to bother writing an entire C program for. Almost every shell script I write now uses AWK for something or other. AWK is definitely a must-learn tool. To learn more about AWK, read *The AWK Programming Language* by Aho, Kernighan, and Weinberger (Addison Wesley, 1988).



Why AWK?

There are a number of reasons for choosing AWK. First and foremost, AWK has direct support for regular expression matching. Because my technique for rule-based gender detection relies so heavily on regular expressions, AWK is the natural language choice. Because every line in an AWK program is fundamentally an *if...then...else* construct, rules can be translated to AWK almost verbatim.

Second, because AWK programs have so little overhead, it was easy to just start writing the program. The minimal overhead also let me keep the code short enough to see the whole program on screen at once.

Third, AWK comes free with almost every UNIX system and workalike. So if you're using UNIX, you almost certainly have an implementation of AWK. Free versions are also available for DOS platforms. Check your local bulletin board system or shareware and public domain warehouse for one.

Fourth, AWK is an interpretive language, so it provides quick turnaround. Because so much trial and error is involved in forging quality rules, it helps not to have to wait for code to compile.

Fifth, once the code is in its final form (when you get tired of trying to find new names that cause it to fail), you can compile it using one of the commercially available AWK compilers or by converting it to C and compiling it. Many C compilers come with a function called *regexp()*, which tells you if a given regular expression matches a given string. Using *regexp()*, it's not a difficult task to convert gender-guessing code from AWK to C.

Combining my technique with a more traditional one has further advantages. Straight text matching is a lot faster than pattern matching, especially if you use a hash function on the data. Then, you can store all the really common names, like John, David, and Jennifer, in a hash table. Only if a name is not present in the hash table would you use the pattern matcher.

Beyond gender guessing

But where do we go from here? Does rule-based programming have any use beyond guessing genders? Fortunately, it does. For example, you can write a spell-checker that uses patterns to look for misspellings, employing rules like *^.*[^g]ht.** → *misspelled*. (You'll need an exception for bathtub, though.)

You can do phonetic text comparisons with a rule-based system. The idea is to map every token to an equivalent phonetic spelling. So you might convert Smith and Smythe to something like *Smi*δ, where δ represents the soft *th* diphthong. This example is a bit trickier to convert to AWK, but it can still be done. And it can produce more accurate results than SOUNDEX, an algorithm found in a number of database packages.

If you want to write a program to hyphenate English words, you would have to buy an on-line hyphenation dictionary or spend considerable time entering the hyphenations for all 500,000 words in the dictionary yourself. But, by using patterns and the rule-based approach I've outlined in this article, you should be able to write a hyphenating program comparatively quickly and get the added benefit of not needing to store an enormous data file.

In general, the techniques I've presented here are applicable anytime you need to categorize a large but finite set of tokens made up of arbitrary but not random character sequences. I say "not random" because it would be too difficult to produce patterns if any sequence of letters composed a valid name, as opposed to pronounceable sequences of letters.

The next time you think you need to create a giant lookup table to solve a programming problem, it may well be worth using the rule-based approach I've presented instead. Using patterns to match many names at once saves a lot of time and effort typing. It's a lot more fun to program, too. ■

Scott Pakin is a senior math and computer science major at Carnegie Mellon University in Pittsburgh, Pa.

Artwork: Dwight Been